Extensible Node Implementation – Start Small, Design for Growth

When developing a node-based system, it's wise to begin with a small set of core node types and a flexible architecture that can accommodate many more later. In practice, this means implementing just a few essential node classes initially, but structuring the code so that adding new ones is straightforward. This follows the YAGNI principle ("You Aren't Gonna Need It"), which advises against building future capabilities before they're needed . By focusing on immediate requirements (e.g. a handful of basic nodes) and not over-engineering for dozens of hypothetical features, we avoid wasted effort and complexity . At the same time, the design should adhere to the Open-Closed Principle – **open for extension, closed for modification** . In practice, this might involve defining a common Node interface or base class such that new node types can be added (via subclassing or composition) without altering the existing code's logic . For example, if our initial nodes include arithmetic operations and data inputs, we ensure the node framework (graph evaluation, scheduler, etc.) is general enough that introducing, say, 100 specialized nodes down the line only requires writing those new node classes, not rewriting the engine.

In short, implement **only what you need now** (perhaps a few node types) but build the infrastructure to be easily **extensible**. The specific choice of which nodes to start with is not critical – the original stakeholder even noted "I don't really care" regarding the initial set – because what matters more is that the system can grow smoothly. Prioritize a clean, modular design over prematurely adding every possible node. This approach lets you deliver immediate functionality and later incorporate additional nodes as requirements evolve, without breaking existing functionality .

Single Data Type (f64) and Fixed Input Arity for Nodes

The system assumes all numeric data is in double-precision floating point (f64). Standardizing on a single data type simplifies the design and implementation: every node can operate on the same type (no need for generics or multiple numeric types), and integration with libraries like NumPy is seamless. Python's built-in float is a 64-bit double precision number, providing about 15 decimal digits of precision . This is usually sufficient for financial and time-series data analysis, offering a good balance of range and precision. It's also efficient on modern hardware – using doubles allows vectorized operations and avoids performance hits that come from using Python objects for numbers . In fact, libraries like pandas prefer floats (with NaN for missing values) over Python None precisely to maintain efficient numeric arrays . By using f64 everywhere, we ensure consistency and take advantage of fast numerical computation.

Each node is defined to take a fixed number of inputs (0, 1, 2, or more), rather than a variable-length argument list. This means every node class has a predetermined arity. For example, a BestBidPrice node might take 0 other nodes (it's a data source for a given symbol), a Sum node takes exactly 2 inputs (e.g. Sum(bid, ask)), and perhaps a specialized node could take 3 or 4 if needed – but there is no generic "N-input" node that accepts an arbitrary list of inputs. Designing nodes with a fixed input count simplifies their interface and usage: the constructor or factory for each node knows exactly what parameters to expect. It also makes the data flow explicit – for instance, seeing Sum(x, y) in code is clearer than a variadic Sum([x,y,...]) and ensures each node's operation is well-defined. While it's possible to chain nodes to achieve an aggregate effect (e.g. summing three signals by doing Sum(a, Sum(b, c))), the framework doesn't require any single node to handle a dynamic number of inputs. This aligns with typical design in dataflow and circuit models, where most components have a fixed number of connections. The result is a simpler implementation with no need to handle edge cases of "missing" or extra inputs for a given node type. Should a truly variadic operation be needed in the future, it can be introduced as a new node class (for example, a SumN node that explicitly takes a list of inputs), keeping in line with the extensibility principle above.

Clear Python API vs. Operator Overloading

The Python API for building the node graph emphasizes **clarity and explicitness** over clever operator overloads. This means using descriptive classes or functions for combining nodes (like calling Sum(node1, node2) or ConstantProduct(0.5, some_node)) rather than relying on Python's special methods to override + or *. While Python does allow operator overloading (and some libraries use it to create DSLs, e.g. allowing node1 + node2 to produce a new combined node), the decision here is that it's "not needed" for this system. Keeping the API straightforward makes the code easier to read and understand at a glance – aligning with the Zen of Python aphorisms that *"Explicit is better than implicit"* and *"Readability counts."* . Each operation is an explicit constructor or function call, so there's no hidden magic in what an expression means; a new developer can see Sum(bid_price, ask_price) and immediately know it produces a sum node, whereas a bid_price + ask_price could be misread as a numeric addition unless one is aware of the operator overloading.

Another benefit of avoiding overloaded operators is reducing "surprise" or unintended interactions. Operator overloading in Python can be powerful but is essentially a form of syntactic sugar – a bit of implicit magic. Some developers caution that such magic should be used sparingly and with care . By not overloading operators, the library stays closer to plain Python semantics, which may help with debugging and tooling (e.g. IDEs and linters don't have to guess the meaning of + on custom objects). The API remains clear: to multiply a node by 0.5, you call a ConstantProduct(0.5, node) constructor (or perhaps a method like node.scale(0.5) if provided), making the intent unambiguous. This design choice favors explicitness and maintainability, even if it means slightly more verbose code. The guiding philosophy is that **clarity trumps brevity** for this library's API – a conscious trade-off to make the behavior obvious to users and avoid errors stemming from overloaded operator misuse .

Handling Node Outputs: None vs. Retaining Previous Value

In a node-based system that processes streaming or time-series data, there will be times when a node has no new information to output (for example, if a price hasn't changed since the last tick, the mid-price remains the same). It's important to distinguish between *actively outputting a "null" value* versus *simply not updating because nothing changed*. One idea discussed was allowing a node's update method to **return None** to signal it has no new value at the current step. While this might seem convenient (as a quick way to indicate "no update"), it introduces ambiguity. We must not confuse a None return with a node *declining to update* and holding its previous value. In Python, None is an actual value that could propagate through the system if not handled carefully, potentially polluting computations or being misinterpreted as a legitimate output (e.g. a missing data point). Moreover, mixing None into a numeric workflow forces special-case logic and breaks the efficiency of numeric operations (since a None would change arrays to object dtype, for instance, wrecking vectorization) .

A clearer approach is to handle the "no new output" scenario at the framework level or via node state, rather than using None as a sentinel. For example, each node can store its last output value internally. If during an update cycle the node determines that its inputs haven't changed (and thus its output would remain the same), it can simply leave its output state as-is and perhaps inform the scheduler that nothing changed. In this design, the node's output stays at the previous value without emitting a special marker. The engine could implement a check (dirty flag or timestamp) to see if any upstream input has changed; if not, it can skip recomputing the node at all. This way, *not updating* is an implicit action (do nothing) rather than an explicit None value flowing through. It aligns with patterns in reactive programming – for instance, a "distinct until changed" strategy where identical consecutive values are not re-emitted to downstream consumers (thus the system naturally holds the last value until a real change occurs).

By avoiding None in the output stream, we ensure that all values traveling through the nodes are actual data (floats in this case) or at least consistently typed. If truly no value is available initially, one could model that as a special node state (like "uninitialized") or use NaN for missing numeric data, but not conflate it with the control logic of skipping updates. The key point is that **a node retaining its previous value should not be represented as a None output**, because that would falsely suggest the node's value *is now "nothing"* rather than "unchanged".

Using None as a control signal can easily be misinterpreted by other parts of the code or by future developers. Thus, the design choice is: if a node has no new result, it simply continues to report its last known result (no function return needed to indicate this, or perhaps a method returns an explicit flag separate from the value). This approach prevents confusion between *“no update”* and *“null value”* and keeps the data flow consistent. In summary, **skip producing new output rather than outputting a None** – the system will treat the node’s value as persisting from the previous cycle. This avoids accidental resets or type problems and makes the temporal behavior of the node graph clear (values change only when there are actual updates).